Comp Sec Activity 5

Exercise
Issuing the following command.

openssl s_client -connect twitter.com:443

Once connected, you may try
GET / HTTP/1.0
[Enter twice]
(Note that the server may return HTTP 404. This is completely normal since we did not send a request for a valid resource.)
Repeat the same step again, now with

openssl s_client -connect twitter.com:443 -CAfile ca-certificates.crt

This command basically connects to port 443 (HTTPS) with the TLS/SSL. This is like a standard telnet command, but with openssl performing the encryption for you.

1. From the two given openssl commands, what is the difference?

Note: If your operating system does not show any error in the first command, try

openssl s_client -connect twitter.com:443 -CApath empty_dir.

If the results are still the same, your system is not reliable. You may ignore this

Answer

openssl s_client -connect twitter.com:443

This command connects to Twitter over port 443 and performs a full TLS handshake (ClientHello, ServerHello, certificate exchange, and key setup). It doesn't specify any CA file, so verification depends on the OS trust store. On macOS, OpenSSL uses the system keychain, so the certificate was trusted and

returned code 0 (ok).

openssl s_client -connect twitter.com:443 -CAfile ca-certificates.crt

This also performs the TLS handshake but explicitly provides a CA bundle for certificate verification. OpenSSL validates Twitter's certificate chain against the given file, confirming it's signed by a trusted CA. Verification succeeded and

returned code 0 (ok).

openssl s_client -connect twitter.com:443 -CApath empty_dir.

The TLS handshake still completes normally, but the specified CA directory is empty, leaving no trusted roots for verification. Although encryption works, OpenSSL cannot verify the server's identity, resulting in

code 20 (unable to get local issuer certificate).

Differences:

They differ in **how the server's certificate is verified**.

- The first command relies on the operating system's built-in trust store (e.g., macOS keychain)
- The second explicitly uses a provided CA file (ca-certificates.crt) for verification
- The third has no trusted CAs because the specified directory is empty—so the handshake succeeds but verification fails with code 20.

2. What does the error (verify error) in the first command mean? Please explain.

Answer

After
openssl s_client -connect twitter.com:443 -CApath empty_dir.
Return
Verify return code: 20 (unable to get local issuer certificate)

The verify error means that OpenSSL successfully completed the TLS handshake and received the server's certificate, but could not verify it because no trusted Certificate Authority (CA) was found in the specified path (empty_dir). Therefore, it shows verify error: unable to get local issuer certificate, indicating that the certificate chain could not be validated even though the connection itself was established.

3. Copy the server certificate (beginning with -----BEGIN CERTIFICATE----- and ending with -----END CERTIFICATE-----) and store it as twitter_com.cert. Use the command openssl x509 -in twitter_com.cert -text to show a text representation of the certificate content. Briefly explain what is stored in an X.509 certificate (i.e. data in each field).

Answer

```
› openssl x509 -in twitter_com.cert -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            06:53:96:80:57:c6:c6:dc:f5:31:36:db:c5:c4:9f:bc:9e:a8
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: C=US, O=Let's Encrypt, CN=E6
        Validity
            Not Before: Aug 19 20:42:10 2025 GMT
            Not After : Nov 17 20:42:09 2025 GMT
        Subject: CN=twitter.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:ba:5c:dc:0e:2c:0f:e4:16:59:54:06:42:42:c0:
                    8a:7c:a1:8c:cd:1e:46:89:76:88:6a:ea:dd:41:8e:
                    3b:aa:6e:3b:d3:48:2a:a2:e4:51:97:d9:d0:b3:52:
                    a9:3e:d7:4b:2e:d0:cf:27:64:0e:f7:7c:77:b3:cf:
                    59:f5:51:03:a6
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                7B:A9:AE:D2:07:70:3A:3D:71:04:96:D1:F1:1F:F9:B0:4B:6E:DC:F1
            X509v3 Authority Key Identifier:
                93:27:46:98:03:A9:51:68:8E:98:D6:C4:42:48:DB:23:BF:58:94:D2
            Authority Information Access:
                CA Issuers - URI:http://e6.i.lencr.org/
            X509v3 Subject Alternative Name:
                DNS:*.twitter.com, DNS:cdn.syndication.twitter.com, DNS:twitter.com
            X509v3 Certificate Policies:
                Policy: 2.23.140.1.2.1
            X509v3 CRL Distribution Points:
                Full Name:
                  URI:http://e6.c.lencr.org/41.crl

            CT Precertificate SCTs:
                Signed Certificate Timestamp:
                    Version   : v1 (0x0)
                    Log ID    : CC:FB:0F:6A:85:71:09:65:FE:95:9B:53:CE:E9:B2:7C:
                                22:E9:85:5C:0D:97:8D:B6:A9:7E:54:C0:FE:4C:0D:B0
                    Timestamp : Aug 19 21:40:40.445 2025 GMT
                    Extensions: none
                    Signature : ecdsa-with-SHA256
                                30:45:02:21:00:F5:28:54:C5:A8:E1:8A:DB:77:A5:BA:
```

An X.509 certificate contains structured information used to identify and verify a server's public key. The main fields include:

- **Version:** Indicates the X.509 standard version (v3).
- **Serial Number:** A unique ID assigned by the certificate authority (CA).
- **Signature Algorithm:** The cryptographic algorithm used for signing (e.g., ECDSA with SHA-384).
- **Issuer:** The organization that issued the certificate (e.g., Let's Encrypt).
- **Validity:** The time period during which the certificate is valid ("Not Before" and "Not After").
- **Subject:** The entity the certificate is issued to (e.g., CN=twitter.com).
- **Subject Public Key Info:** The public key and algorithm used (e.g., EC key, P-256 curve).
- **Extensions:** Extra attributes such as Key Usage (Digital Signature), Extended Key Usage (TLS Web Server Authentication), Subject Alternative Name (DNS entries), and Certificate Policies.
- **Signature:** The CA's digital signature used to verify the certificate's authenticity and integrity.

In summary, an X.509 certificate stores the **identity of the subject**, its **public key**, the **issuer's information**, the **validity period**, and **digital signatures and extensions** needed to ensure secure, trusted communication.

4. From the information in exercise 3, is there an intermediate certificate? If yes, what purpose does it serve?

Hint: Look for an issuer and download the intermediate certificate. You may use the command openssl x509 -inform der -in intermediate.cert -text to show the details of the intermediate certificate. (Note that the -inform der is for reading the DER file. The default file format for x509 is the PEM file.)

Answer

Yes, there is an **intermediate certificate**. In Exercise 3, the **Issuer** of the Twitter certificate is C=US, O=Let's Encrypt, CN=E6, while the **Subject** is CN=twitter.com. This shows that the Twitter server certificate was not signed directly by a root CA but by an **intermediate CA (E6)** issued by Let's Encrypt.

The purpose of the intermediate certificate is to **form a trust chain** between the root CA and the server certificate. It acts as a bridge — the **root CA** signs the **intermediate certificate**, and the **intermediate** signs the **server certificate**. This improves security because the root CA's private key is kept offline, while the intermediate CA handles daily signing operations. In summary, the intermediate certificate ensures **trusted verification** without exposing the root CA key.

5. Is there an intermediate CA, i.e. is there more than one organization involved in the certification? Say why you think so.

Answer

Yes, there is an **intermediate CA**, meaning more than one organization is involved in the certification process. The **Issuer** of the Twitter certificate is Let's Encrypt E6, while the root CA (that ultimately signs E6) belongs to another higher-level trusted organization. This shows a **chain of trust** where the **root CA** certifies the **intermediate CA (Let's Encrypt)**, and the intermediate CA certifies the **server certificate (twitter.com)**. This multi-level structure increases security and reliability by keeping the root CA's key offline and delegating certificate issuance to the intermediate CA.

6. What is the role of ca-certificates.crt?

Answer

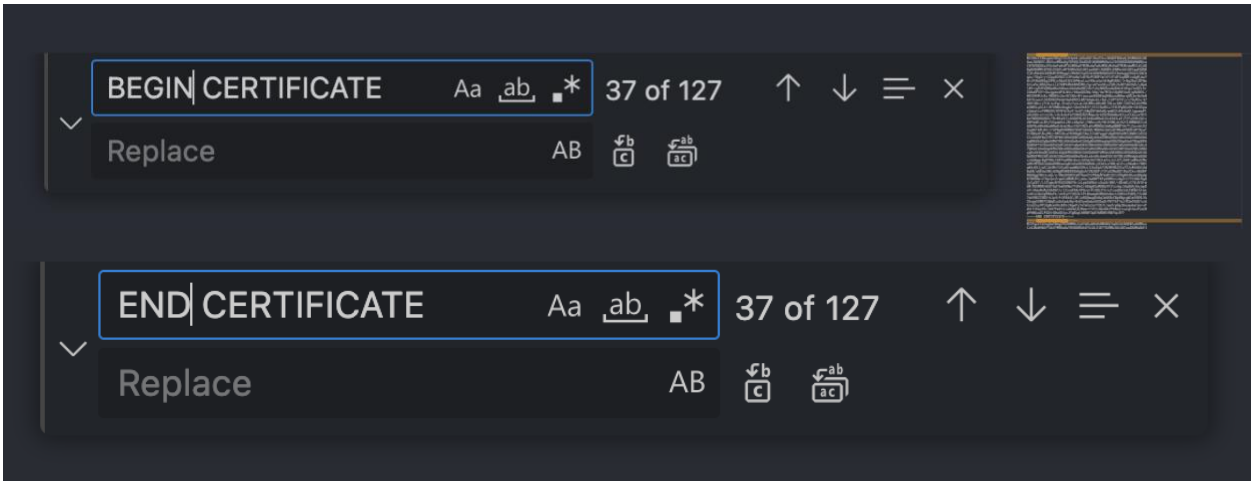
The file **ca-certificates.crt** is a **bundle of trusted root Certificate Authority (CA) certificates** used by the operating system or OpenSSL to verify the authenticity of server certificates during a TLS/SSL handshake. When a server presents its certificate, OpenSSL compares the issuer information against the list of trusted Cas in this file to confirm that the certificate was signed by a recognized and trusted authority. In short, ca-certificates.crt acts as the **trust anchor** that allows OpenSSL (and other applications) to validate certificate chains and ensure secure, verified HTTPS connections.

7. Explore the ca-certificates.crt. How many certificates are in there? Give the command/method you have used to count.

Answer

There are **127 certificates**.

Use Find **(Control + F)** to find **BEGIN CERTIFICATE** and **END CERTIFICATE** to cross check eachother for count.



8. Extract a root certificate from ca-certificates.crt. Use the openssl command to explore the details. Do you see any Issuer information? Please compare it to the details of twitter's certificate and the details of the intermediate certificate.

Answer

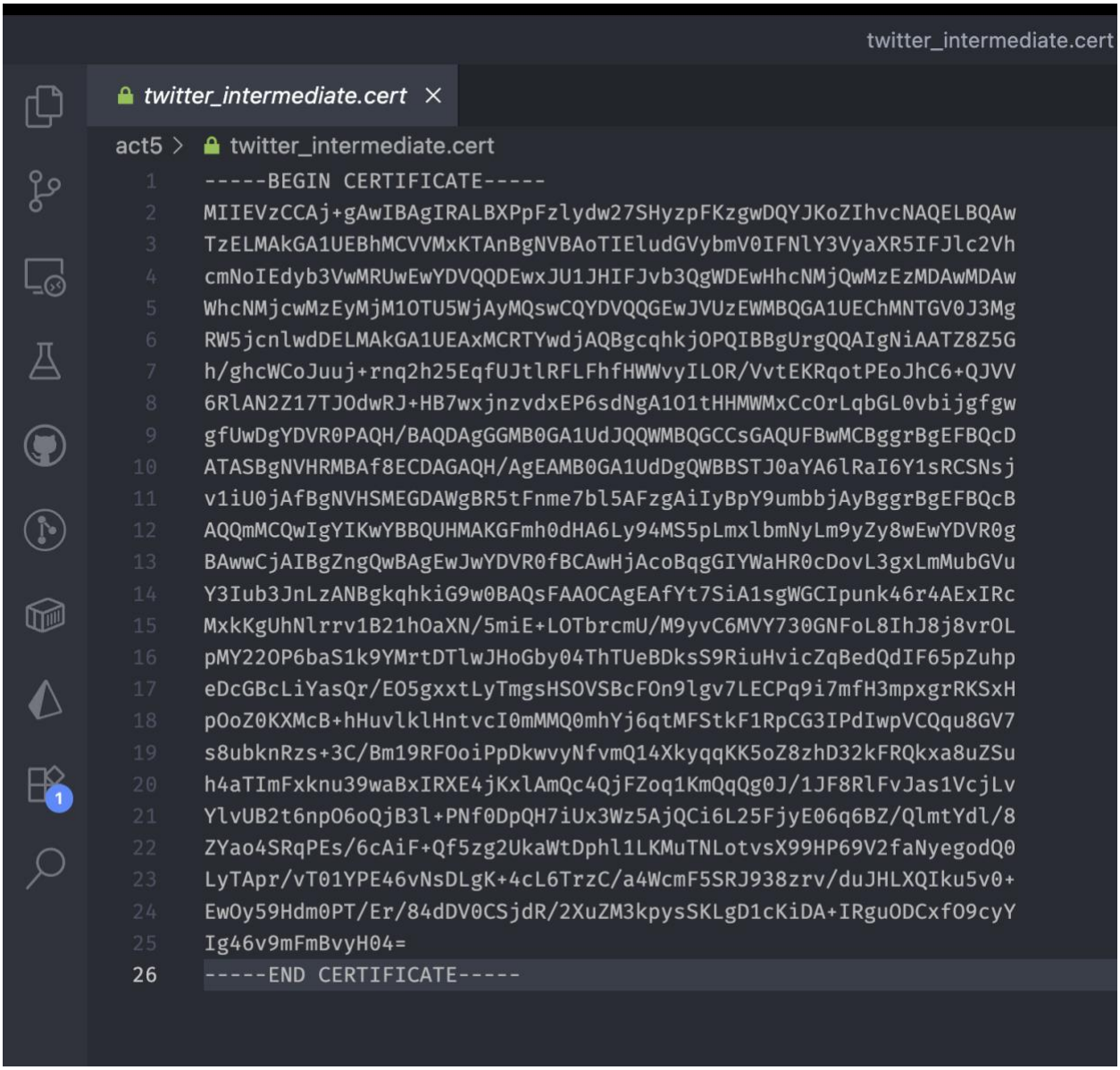
- **openssl x509 -in ca-certificates.crt -noout -subject -issuer**
  - subject=CN=ACCVRAIZ1, OU=PKIACCV, O=ACCV, C=ES
  - issuer=CN=ACCVRAIZ1, OU=PKIACCV, O=ACCV, C=ES
- **openssl x509 -in twitter_com.cert -noout -subject -issuer**
  - subject=CN=twitter.com
  - issuer=C=US, O=Let's Encrypt, CN=E6
- **openssl x509 -in twitter_intermediate.cert -noout -subject -issuer**
  - subject=C=US, O=Let's Encrypt, CN=E6
  - issuer=C=US, O=Internet Security Research Group, CN=ISRG Root X1

Conclusion:

- Each certificate is **signed by the one above it**, creating a verification chain: **ISRG Root X1 (root)** → **Let's Encrypt E6 (intermediate)** → **twitter.com (server)**.
- The **root CA** is self-signed and trusted by your operating system (via ca-certificates.crt).
- The **intermediate CA** acts as a bridge that signs end-entity certificates like Twitter's, while keeping the root key offline.

9. If the intermediate certificate is not in a PEM format (text readable), use the command to convert a DER file (.crt .cer .der) to PEM file. openssl x509 -inform der -in certificate.cer -out certificate.pem. (You need the pem file for exercise 10.)

Answer


twitter_intermediate.cert
twitter_intermediate.cert
act5 > twitter_intermediate.cert
-----BEGIN CERTIFICATE-----
MIIEVzCCAj+gAwIBAgIRALBXPpFzlydw27SHyzpFKzgwDQYJKoZIhvcNAQELBQAw
TzELMAkGA1UEBhMCVVMxKTAnBgNVBAoTIEludGVybmV0IFNlY3VyaXR5IFJlc2Vh
cmNoIEdyb3VwMRUwEwYDVQQDEwxJU1JHIFJvb3QgWDEwHhcNMjQwMzEzMDAwMDAw
WhcNMjcwMzEyMjM1OTU5WjAyMQswCQYDVQQGEwJVUzEWMBQGA1UEChMNTGV0J3Mg
RW5jcnlwdDELMAkGA1UEAxMCRTYwdjAQBgcqhkjOPQIBBgUrgQQAIgNiAATZ8Z5G
h/ghcWCoJuuj+rnq2h25EqfUJtlRFLFhfHWWvyILOR/VvtEKRqotPEoJhC6+QJVV
6RlAN2Z17TJOdwRJ+HB7wxjnzvdxEP6sdNgA1O1tHHMWMxCcOrLqbGL0vbijgfgw
gfUwDgYDVR0PAQH/BAQDAgGGMB0GA1UdJQQWMBQGCCsGAQUFBwMCBggrBgEFBQcD
ATASBgNVHRMBAf8ECDAGAQH/AgEAMB0GA1UdDgQWBBSTJ0aYA6lRaI6Y1sRCSNsj
v1iU0jAfBgNVHSMEGDAWgBR5tFnme7bl5AFzgAiIyBpY9umbbjAyBggrBgEFBQcB
AQQmMCQwIgYIKwYBBQUHMAKGFmh0dHA6Ly94MS5pLmxlbmNyLm9yZy8wEwYDVR0g
BAwwCjAIBgZngQwBAgEwJwYDVR0fBCAwHjAcoBqgGIYWaHR0cDovL3gxLmMubGVu
Y3Iub3JnLzANBgkqhkiG9w0BAQsFAAOCAgEAfYt7SiA1sgWGCIpunk46r4AExIRc
MxkKgUhNlrrv1B21hOaXN/5miE+LOTbrcmU/M9yvC6MVY730GNFoL8IhJ8j8vrOL
pMY22OP6baS1k9YMrtDTlwJHoGby04ThTUeBDksS9RiuHvicZqBedQdIF65pZuhp
eDcGBcLiYasQr/EO5gxxtLyTmgsHSOVSBcFOn9lgv7LECPq9i7mfH3mpxgrRKSxH
pOoZ0KXMcB+hHuvlklHntvcI0mMMQ0mhYj6qtMFStkF1RpCG3IPdIwpVCQqu8GV7
s8ubknRzs+3C/Bm19RFOoiPpDkwvyNfvmQ14XkyqqKK5oZ8zhD32kFRQkxa8uZSu
h4aTImFxknu39waBxIRXE4jKxlAmQc4QjFZoq1KmQqQg0J/1JF8RlFvJas1VcjLv
YlvUB2t6npO6oQjB3l+PNf0DpQH7iUx3Wz5AjQCi6L25FjyE06q6BZ/QlmtYdl/8
ZYao4SRqPEs/6cAiF+Qf5zg2UkaWtDphl1LKMuTNLotvsX99HP69V2faNyegodQ0
LyTApr/vT01YPE46vNsDLgK+4cL6TrzC/a4WcmF5SRJ938zrv/duJHLXQIku5v0+
EwOy59Hdm0PT/Er/84dDV0CSjdR/2XuZM3kpysSKLgD1cKiDA+IRguODCxfO9cyY
Ig46v9mFmBvyH04=
-----END CERTIFICATE-----

10. From the given python code, implement the certificate validation.

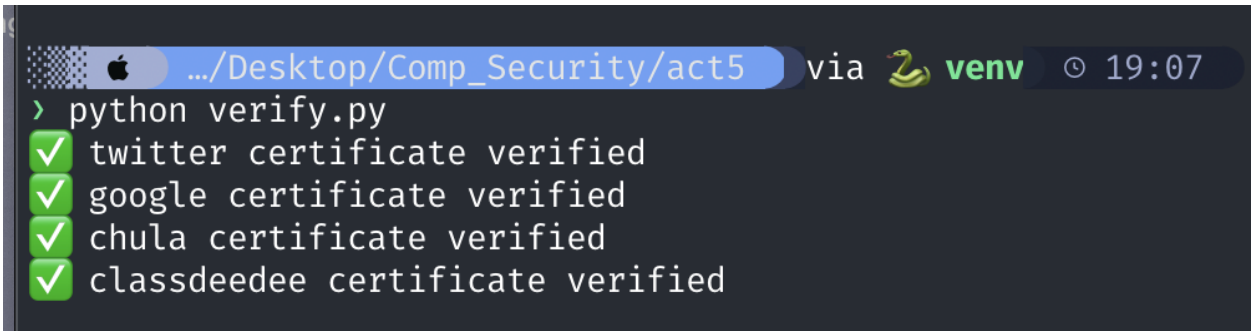
Use your program to verify the certificates of: Twitter, google, www.chula.ac.th, classdeedee.cloud.cp.eng.chula.ac.th

```
from OpenSSL import crypto
import pem

def verify(cert_file_name, int_cert_file_name, title):
    with open(cert_file_name, 'r') as cert_file:
        cert = cert_file.read()
    with open(int_cert_file_name, 'r') as int_cert_file:
        int_cert = int_cert_file.read()
    pems = pem.parse_file('./ca-certificates.crt')
    trusted_certs = []
    for mypem in pems:
        trusted_certs.append(str(mypem))
    trusted_certs.append(int_cert)
    verified = verify_chain_of_trust(cert, trusted_certs)
    if verified:
        print(f'✅ {title} certificate verified')

def verify_chain_of_trust(cert_pem, trusted_cert_pems):
    certificate = crypto.load_certificate(crypto.FILETYPE_PEM, cert_pem)
    # Create and fill a X509Store with trusted certs
    store = crypto.X509Store()
    for trusted_cert_pem in trusted_cert_pems:
        trusted_cert = crypto.load_certificate(crypto.FILETYPE_PEM,
                                               trusted_cert_pem)
        store.add_cert(trusted_cert)
    # Create a X590StoreContext with the cert and trusted certs
    # and verify the the chain of trust
    store_ctx = crypto.X509StoreContext(store, certificate)
    # Returns None if certificate can be validated
    result = store_ctx.verify_certificate()
    if result is None:
        return True
    else:
        return False

verify('twitter.cert', 'twitter_intermediate.cert', 'twitter')
verify('google.cert', 'google_intermediate.cert', 'google')
verify('chula.cert', 'chula_intermediate.cert', 'chula')
verify('classdeedee.cert', 'classdeedee_intermediate.cert', 'classdeedee')
```



11. Nowaday, there are root certificates for class 1 and class 3. What uses would a class 1 signed certificate have that a class 3 doesn't, and vice versa?

Answer

A **Class 1 certificate** is used for **basic identity validation**, such as verifying only that the email address or domain exists. It's suitable for **personal use, email signing, or low-risk websites** where minimal identity assurance is acceptable.

A **Class 3 certificate**, on the other hand, requires **stronger authentication** — verifying the organization's legal identity and ownership. It is used for **commercial, financial, or e-commerce websites** where users must trust the organization's legitimacy.

In summary, **Class 1 = basic, low-assurance**, while **Class 3 = high-assurance, organization-validated** certificates used in business-critical or sensitive transactions.

12. Assuming that a Root CA in your root store is hacked and under the control of an attacker, and this is not noticed by anyone for months.

a. What further attacks can the attacker stage? Draw a possible attack setup.

If a Root CA is compromised, the attacker can act as a **fake trusted CA** and issue valid certificates for any website. One likely attack is a **phishing or fake-website impersonation**: the attacker creates a counterfeit site (for example, https://bank-login.fake.com) and issues a forged certificate for the real domain bank.com. Because the

certificate appears valid and is signed by a trusted root, users' browsers will show the padlock and accept the connection. This allows the attacker to intercept or steal login credentials, session cookies, and personal data through a **man-in-the-middle (MITM)** attack.

The attacker uses the stolen root key to generate a certificate that seems legitimate for any domain. Since browsers inherently trust the root CA, users receive no warning and may unknowingly send confidential data to the attacker.

b. In the attack you have described above, can we rely on CRLs or OCSP for protection? Please explain

Answer

No, CRLs and OCSP cannot reliably protect against a root CA compromise because both mechanisms depend on the CA's integrity. If the attacker controls the Root CA, they can issue fake certificates and also generate false OCSP "good" responses or omit entries from CRLs. Since browsers typically trust the CA's responses and often soft-fail when OCSP checks are unreachable, users would still accept the forged certificates as valid. Therefore, once a Root CA is compromised, the only effective defense is to remove or distrust that CA from all trusted root stores.